Input/Output (I/O)

CS 3410: Computer System Organization & Programming

Spring 2025

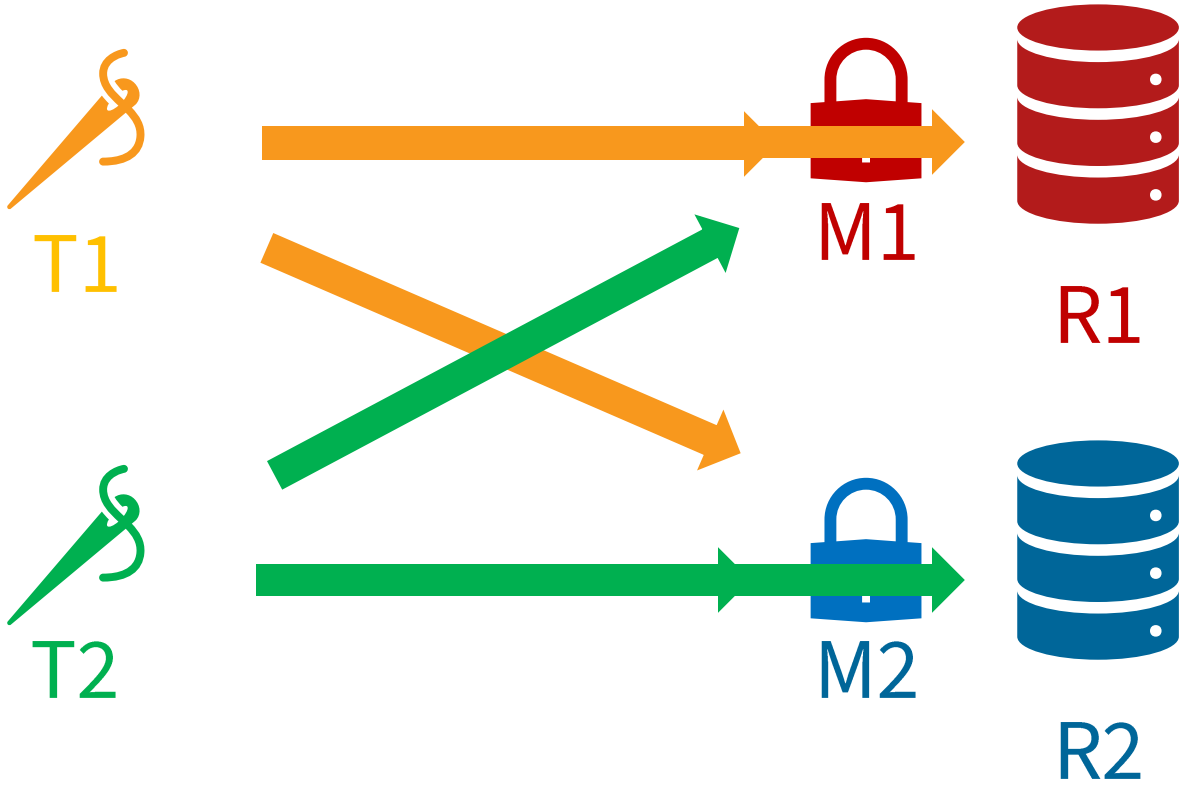
Outline for Today

- **Recap:** Deadlocks & How to Prevent Them
- What are I/O devices?
- How do we connect to I/O devices?
- How do we communicate with I/O devices?

Deadlock

Deadlock occurs when two different threads get stuck waiting for each other.

Both T1 and T2 need to access R1 and R2.



Deadlock using pthreads

Avoiding Deadlock

Key Problem: threads acquire locks in different orders

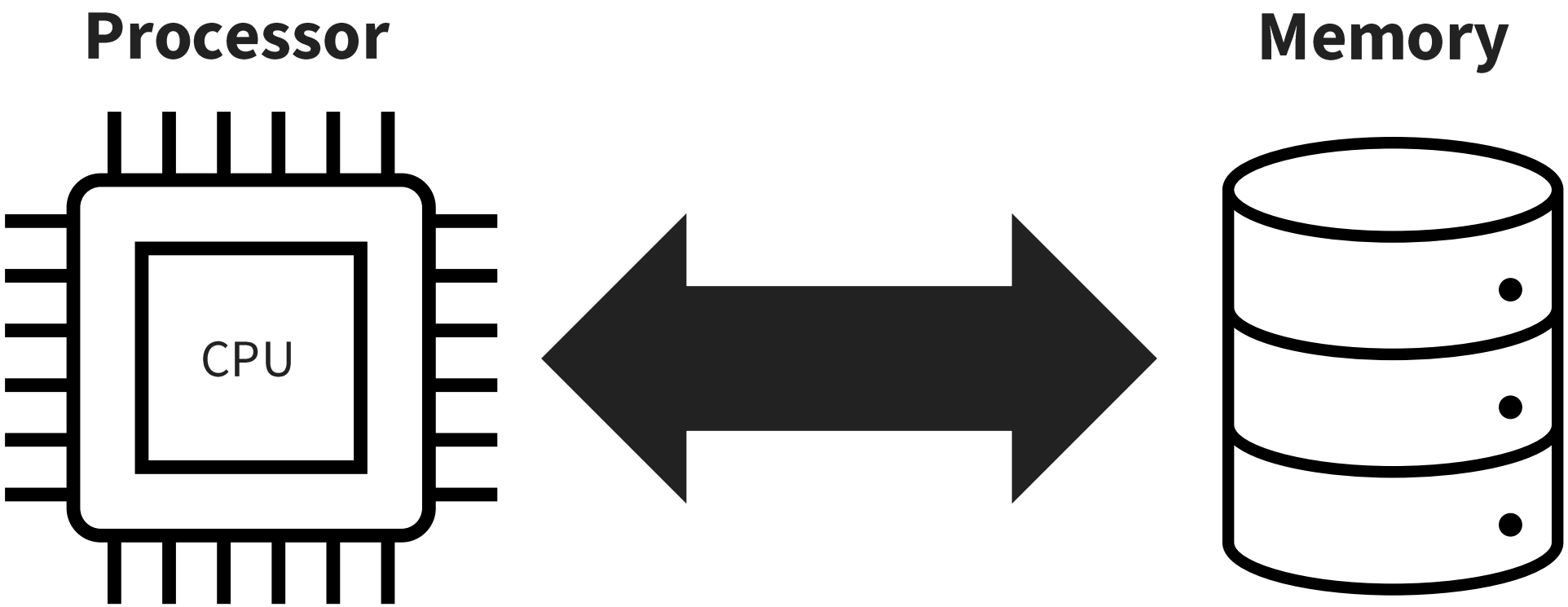
Three Easy Steps to Avoid Deadlock with Mutexes:

1. Decide on a **total order** among all your mutexes
2. Always acquire the mutexes in that order.
3. Always release the mutexes in the *opposite* order.

Think of locks as curly brackets!

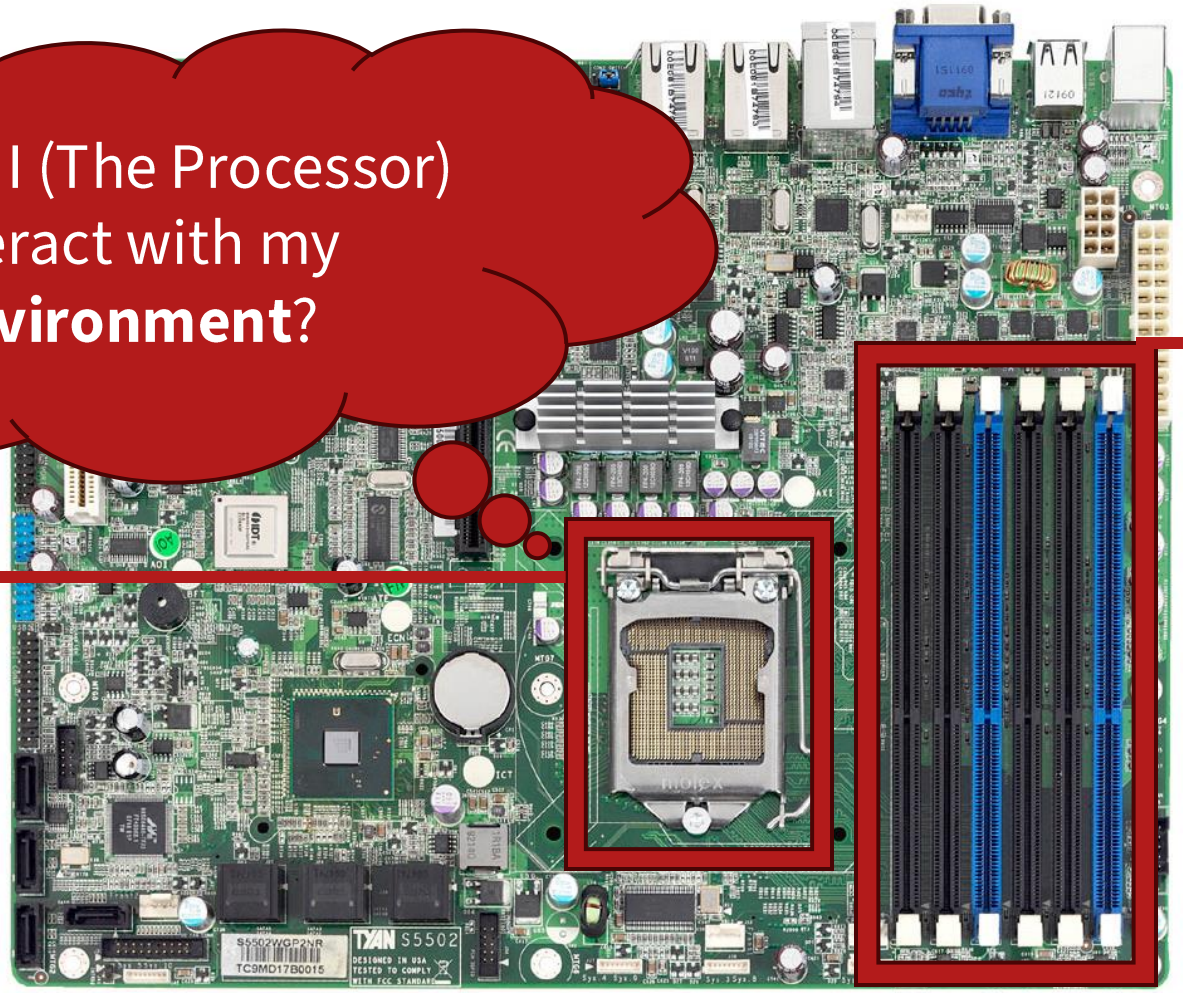
Main Topic: Input/Output (I/O)

Big Picture: Input/Output (I/O)



Big Picture: Input/Output (I/O)

How do I (The Processor) interact with my **environment**?



CPU

Main Memory

SandyBridge Motherboard, 2011 http://news.softpedia.com

Word Cloud: Give an example of an I/O device.

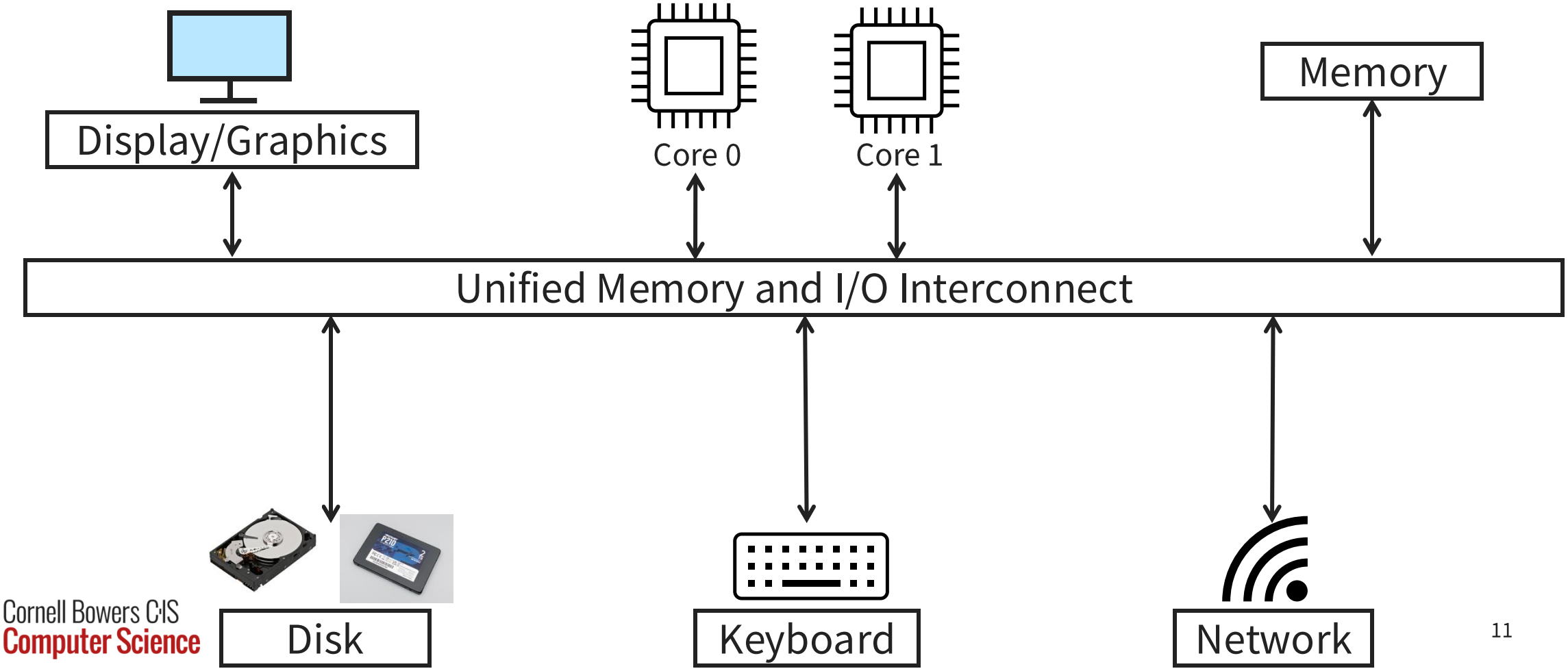
https://PollEv.com/zjs

I/O Devices Enables Interacting with Environment

Device	Behavior	Partner	Data Rate (b/sec)
Keyboard	Input	Human	100
Mouse	Input	Human	3.8k
Sound Input	Input	Machine	3M
Voice Output	Output	Human	264k
Sound Output	Output	Human	8M
Laser Printer	Output	Human	3.2M
Graphics Display	Output	Human	800M – 8G
Network/LAN	Input/Output	Machine	100M – 400G
Network/Wireless LAN	Input/Output	Machine	11M – 10G
Optical Disk	Storage	Machine	5 – 120M
Flash memory	Storage	Machine	32 – 10G
Magnetic Disk	Storage	Machine	800M – 3G

Round 1: All devices on one interconnect

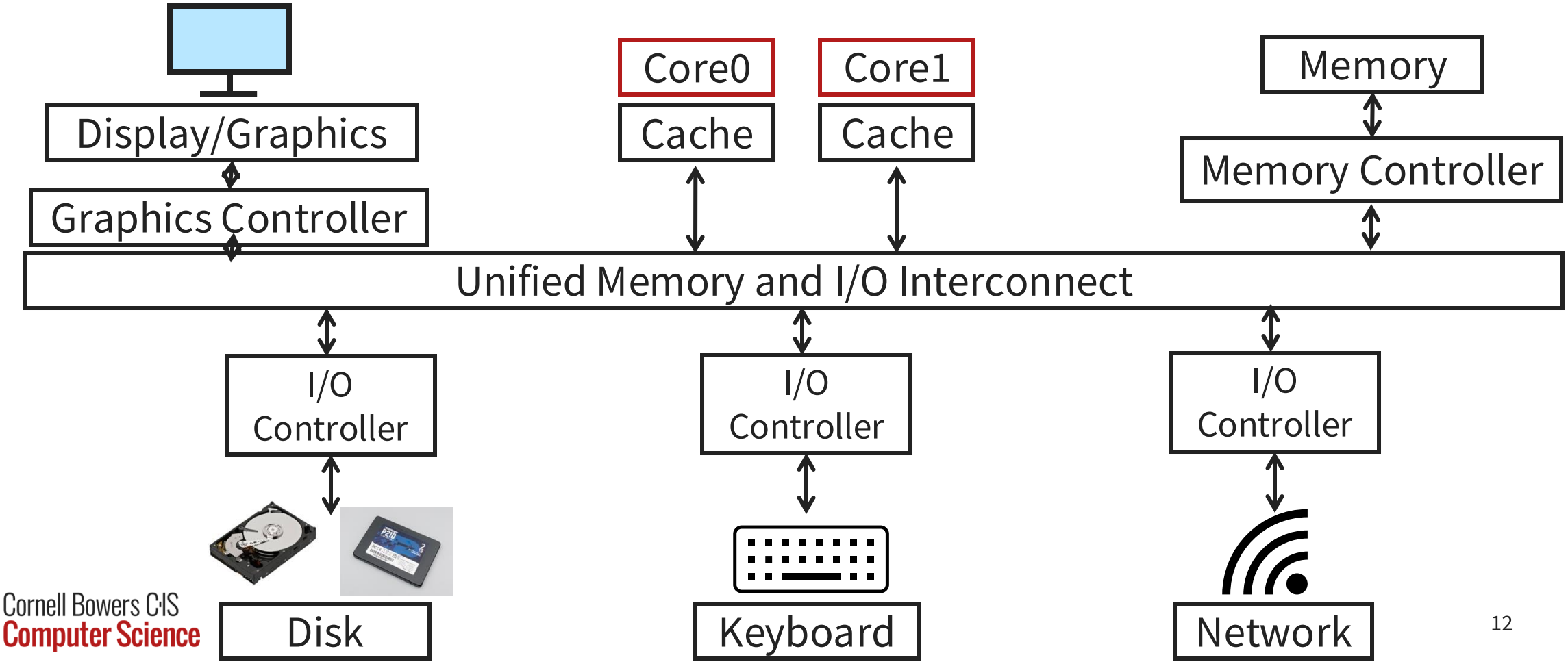
- Replace **all** devices as interconnect changes
- Shared latency across all devices (Main Memory = Keyboard)



Round 2: I/O Controllers

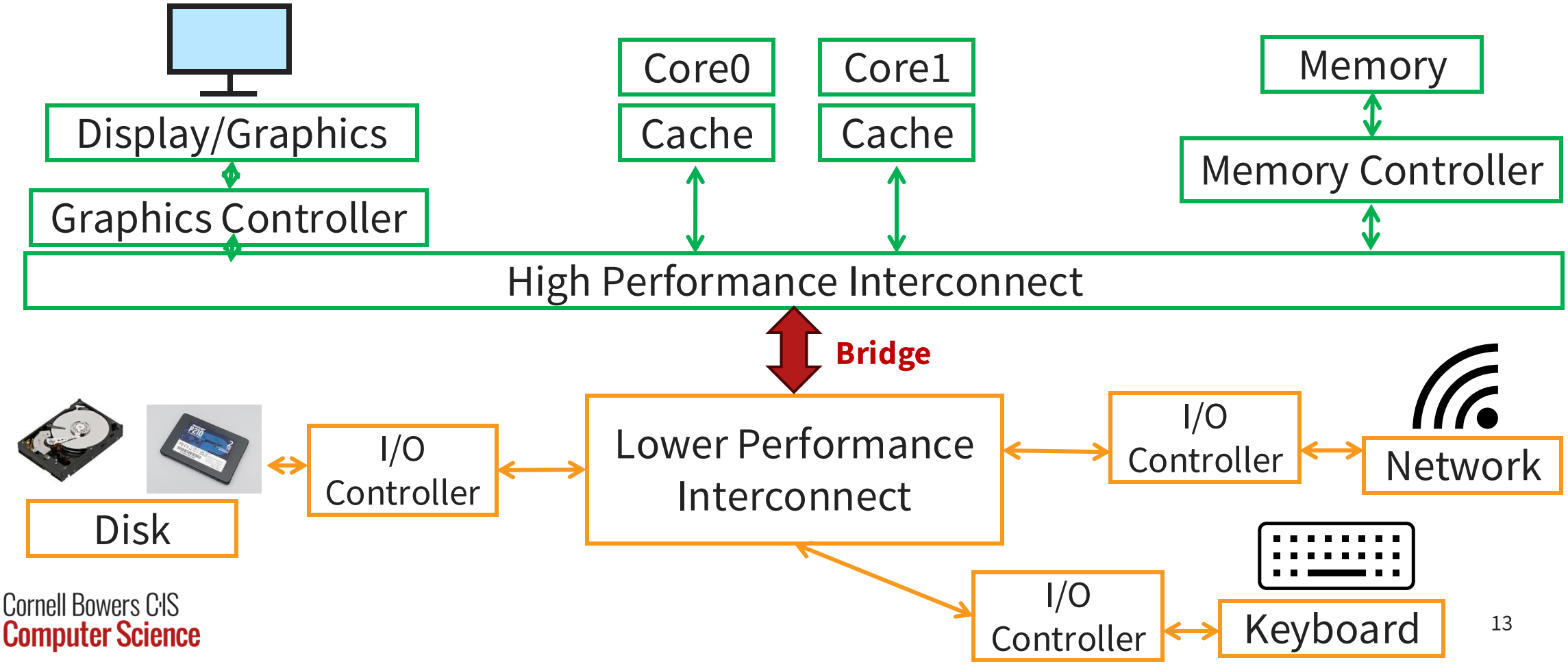
Decouple I/O devices from Interconnect

Enable smarter, more efficient I/O interfaces



Round 3: I/O Controllers + Bridge

Separate high-performance interconnects (e.g., processor, memory, graphics) from lower-performance interconnects (e.g., disk, user input, network)



Bus Types

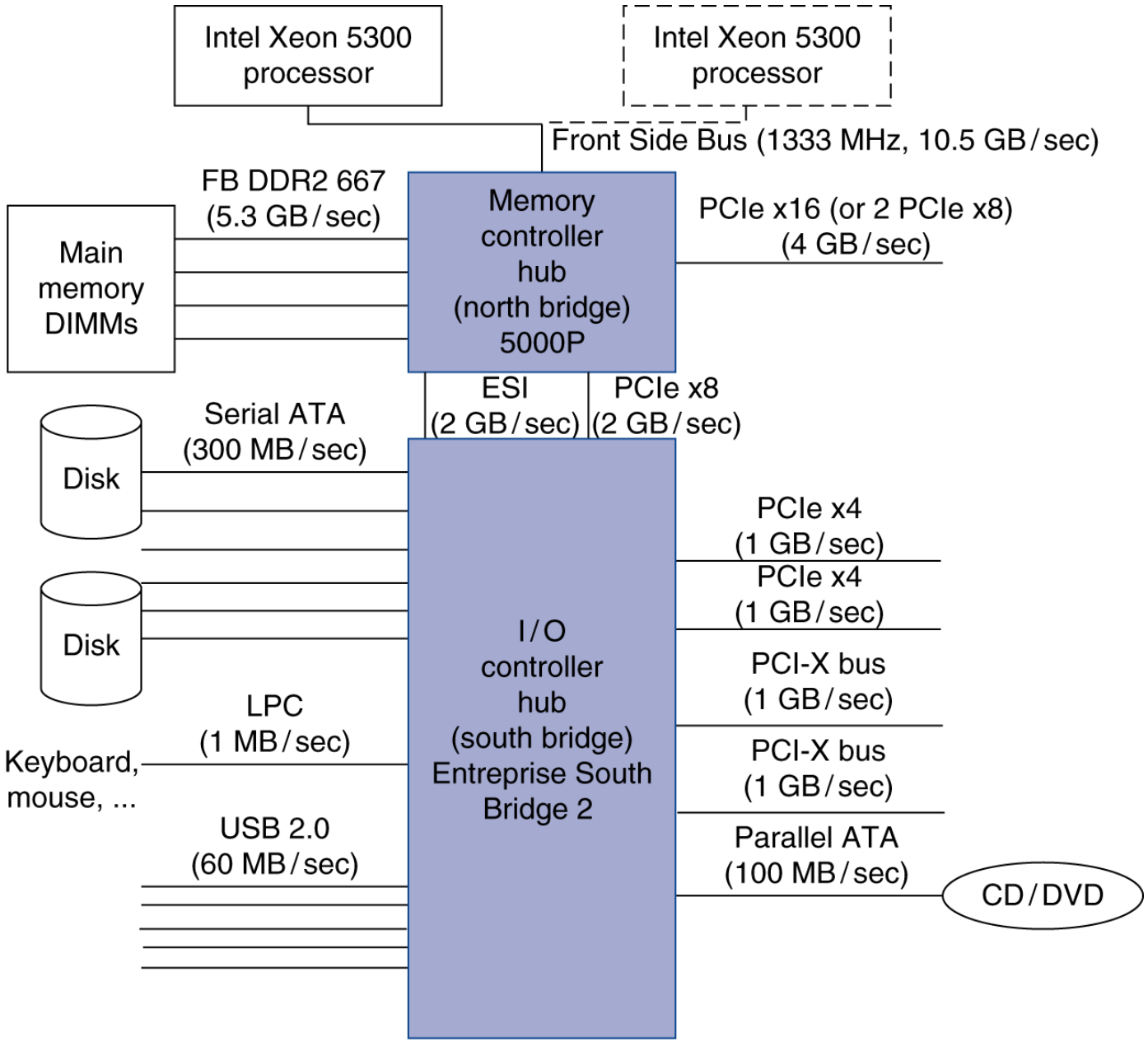
Processor, Memory, Graphics ("Front Side Bus")

- Short, fast, & wide
- Mostly fixed topology, designed as a "chipset"
  - CPU + Caches + Interconnect + Memory Controller + Graphics Controller

I/O and Peripheral busses (PCIe, SATA, SCSI, USB…)

- Longer, slower, & narrower
- Flexible topology, multiple/varied connections
- Interoperability standards for devices
- Connect to processor-memory bus through a bridge
  - Example: Intel's proprietary Direct Media Interface (DMI)

Example


Intel Xeon 5300 processor
Intel Xeon 5300 processor
Front Side Bus (1333 MHz, 10.5 GB/sec)
FB DDR2 667 (5.3 GB/sec)
Main memory DIMMs
Memory controller hub (north bridge) 5000P
PCIe x16 (or 2 PCIe x8) (4 GB/sec)
ESI (2 GB/sec)
PCIe x8 (2 GB/sec)
Serial ATA (300 MB/sec)
Disk
Disk
I/O controller hub (south bridge) Entreprise South Bridge 2
PCIe x4 (1 GB/sec)
PCIe x4 (1 GB/sec)
PCI-X bus (1 GB/sec)
PCI-X bus (1 GB/sec)
Parallel ATA (100 MB/sec)
CD/DVD
LPC (1 MB/sec)
Keyboard, mouse, ...
USB 2.0 (60 MB/sec)

Example Interconnects

Name	Use	Devices per channel	Channel Width	Data Rate (B/sec)
Firewire 800	External	63	4	100M
USB 2.0	External	127	2	60M
USB 3.0	External	127	2	625M
USB 4.0	External	127	2	40G
Thunderbolt 4	External	127	2	80G
Parallel ATA	Internal	1	16	133M
Serial ATA (SATA)	Internal	1	4	300M
PCI 66MHz	Internal	1	32-64	533M
PCI Express v2.x	Internal	1	2-64	16G/dir
…				
PCI Express v6.x	Internal	1	2-64	1T/dir
Hypertransport v2.x	Internal	1	2-64	25G/dir
QuickPath (QPI)	Internal	1	40	12G/dir

Interconnecting Components

Interconnects are (were?) busses

- parallel set of wires for data and control
- shared channel
  - multiple senders/receivers
  - everyone can see all bus transactions
- bus protocol: rules for using the bus wires

Alternative (and increasingly common)

- dedicated point-to-point channels
- Examples: HyperTransport, InfiniBand

Takeaways

Diverse I/O devices require hierarchical interconnects. More recently, these interconnects are starting to more closely resemble point-to-point networks.

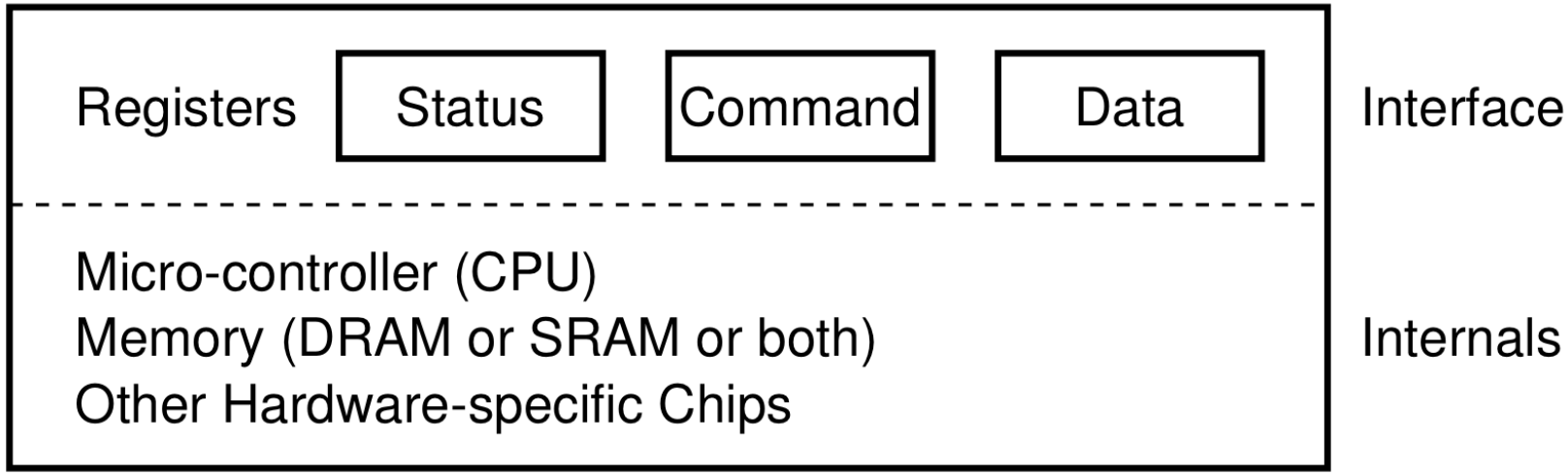
How does the processor interact with I/O devices?

I/O Device Driver Software Interface

```c
// Open a toy " echo " character device
int fd = open("/dev/echo", O_RDWR);

// Write to the device
char write_buf[] = "Hello World!";
write(fd, write_buf, sizeof(write_buf));

// Read from the device
char read_buf [32];
read(fd, read_buf, sizeof(read_buf));

// Close the device
close(fd);

// Verify the result
assert(strcmp(write_buf, read_buf)==0);
```

I/O Device API

All I/O devices use this API!



Typical I/O Device API

- a set of read-only or read/write register

Command registers

- writing causes device to do something

Status registers

reading indicates what device is doing, error codes, etc.

Data registers

Write: transfer data to a device

Read: transfer data from a device

I/O Device API Example

8-bit Status:

PERR	RxTO	TxTO	INH	AL2	SYS	IBF	OBF

Input Buffer Full

Output Buffer Full

8-bit Command:

```
0xAA = “self test”
0xAE = “enable kbd”
0xED = “set LEDs”
```

…

8-bit Data:

Scan code (when reading)
LED state (when writing)

IBM AT Keyboard

How to talk to a device?

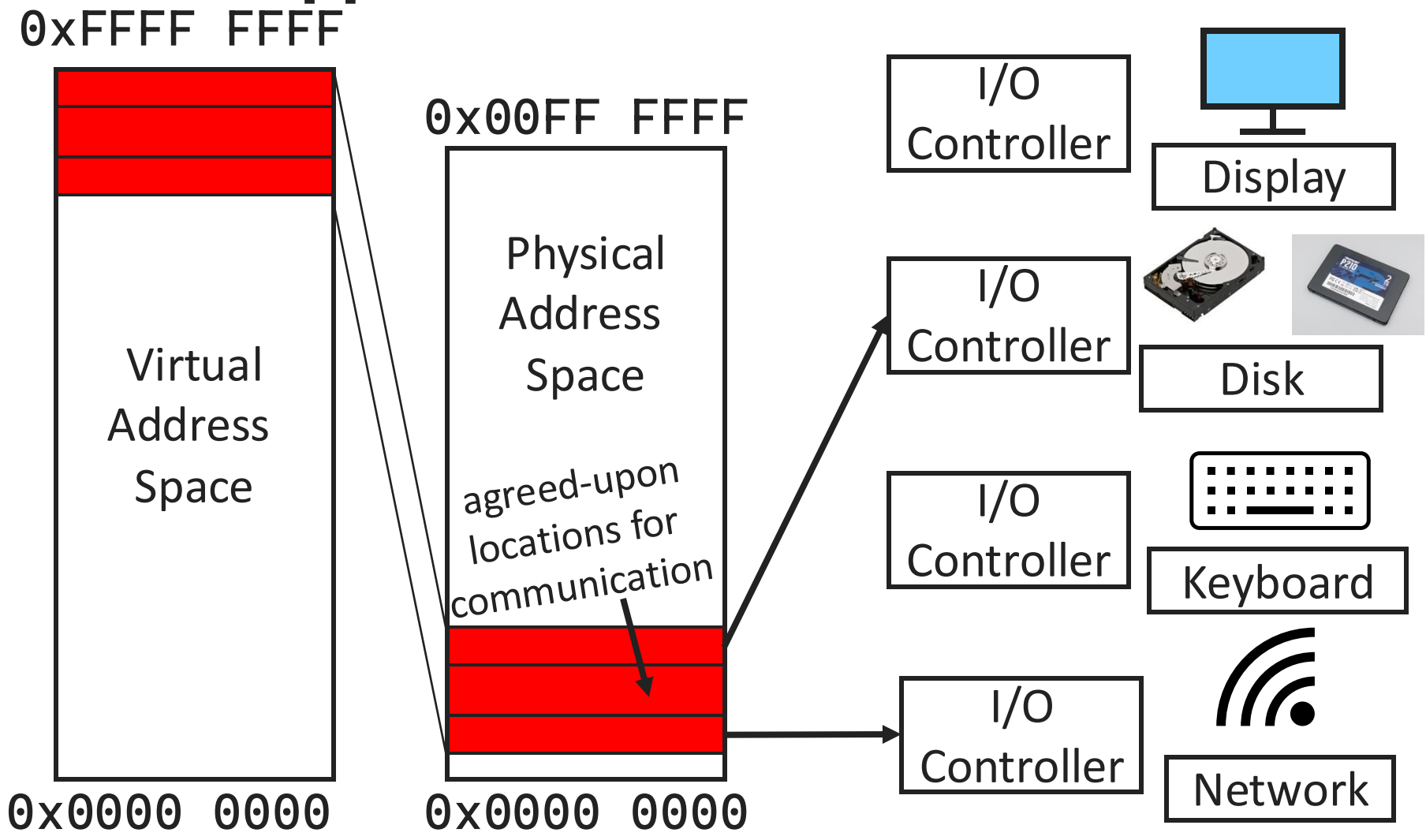
1. **Programmed I/O (PIO)**:

- Special instructions to talk over special busses
- Specify: device, data, direction
    - `inb  $a, 0x64` (keyboard status register)
    - `outb $a, 0x60` (keyboard data register)
- Privileged operation: only allowed in kernel mode (expensive)

2. **Memory-Mapped I/O:**

- Device registers are mapped into virtual address space
    - Accesses to certain addresses redirected to I/O devices
    - Data goes over the memory bus (faster!)
    - Protection: via bits in pagetable entries
- OS+MMU+devices configure mappings
- No new instructions are needed

Memory-Mapped I/O



vs. less-favored alternative = Programmed I/O:

- Syscall instructions that communicate with I/O
- Communicate via special device registers

Device Drivers

Programmed I/O

```
char read_kbd() {
  char status;
  do {
    sleep();
    status = inb(0x64);
  } while(!(status & 1));
  return inb(0x60);
}
```

https://PollEv.com/zjs

syscalls

PollEV Question: Which is better?
(A) Programmed I/O
(B) Memory Mapped I/O
(C) Both have syscalls, both are bad

Memory Mapped I/O

```
struct kbd {
  char status, pad[3];
  char data, pad[3];
};
kbd *k = mmap( ... );

char read_kbd() {
  char status;
  do {
    sleep();
    status = k→status;
  } while(!(status & 1));
  return k→data;
}
```

syscall

I/O Data Transfer

How to talk to device?

- Programmed I/O or Memory-Mapped I/O

How to get events?

- Polling vs. Interrupts

How to transfer lots of data?

```
disk→cmd = READ_4K_SECTOR;
disk→data = 12;
while (!(disk→status & 1) { }
for (i = 0..4k)
buf[i] = disk→data;
```

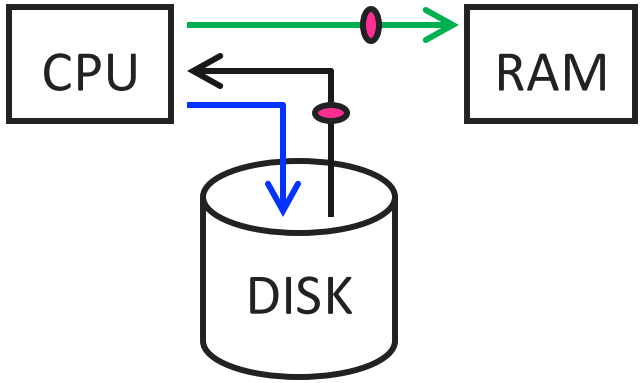
Very,
Very,
Expensive

Data Transfer

1. Programmed: Device ←→ CPU ←→ RAM Transfer

   for (i = 1 .. n)

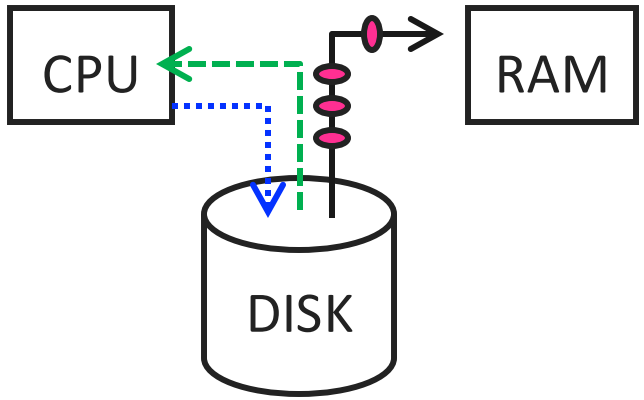
   - CPU issues read request
   - Device puts data on bus & CPU reads into registers
   - CPU writes data to memory



2. Direct Memory Access (DMA): Device ←→ RAM

   - CPU sets up DMA request
   - for (i = 1 … n)

     Device puts data on bus & RAM accepts it
   - Device interrupts CPU after done



DMA Example

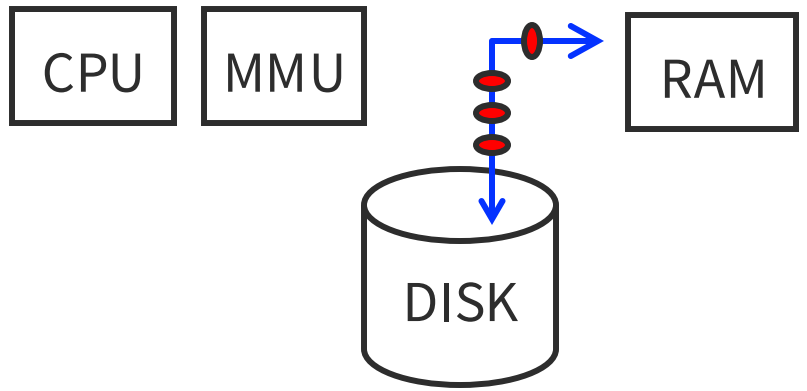
DMA example: reading from audio (mic) input

- DMA engine on audio device… or I/O controller … or …

```
int dma_size = 4*PAGE_SIZE;
int *buf = alloc_dma(dma_size);
...
dev->mic_dma_baseaddr = (int)buf;
dev->mic_dma_count = dma_len;
dev->cmd = DEV_MIC_INPUT |
DEV_INTERRUPT_ENABLE | DEV_DMA_ENABLE;
```

DMA Issues (1): Addressing

Issue #1: DMA meets Virtual Memory
RAM: physical addresses
Programs: virtual addresses



Solution: DMA uses virtual addresses

- OS sets up mappings on a mini-TLB

DMA Example

DMA example: reading from audio (mic) input

- DMA engine on audio device… or I/O controller … or …

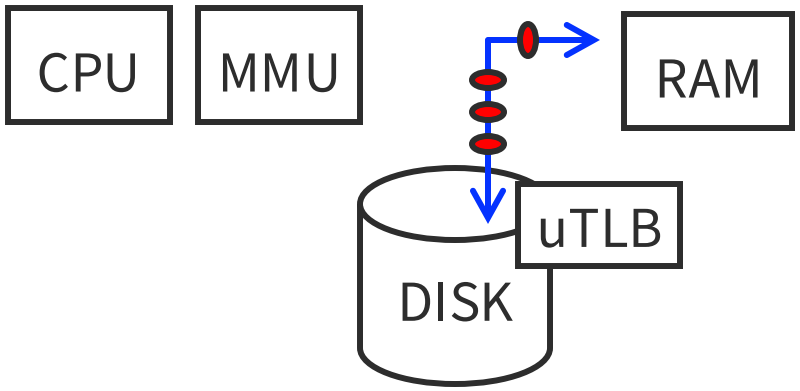
```
int dma_size = 4*PAGE_SIZE;
void *buf = alloc_dma(dma_size);
...
dev->mic_dma_baseaddr = virt_to_phys(buf);
dev->mic_dma_count = dma_len;
dev->cmd = DEV_MIC_INPUT |
DEV_INTERRUPT_ENABLE | DEV_DMA_ENABLE;
```

DMA Issues (1): Addressing

Issue #1: DMA meets Virtual Memory

RAM: physical addresses

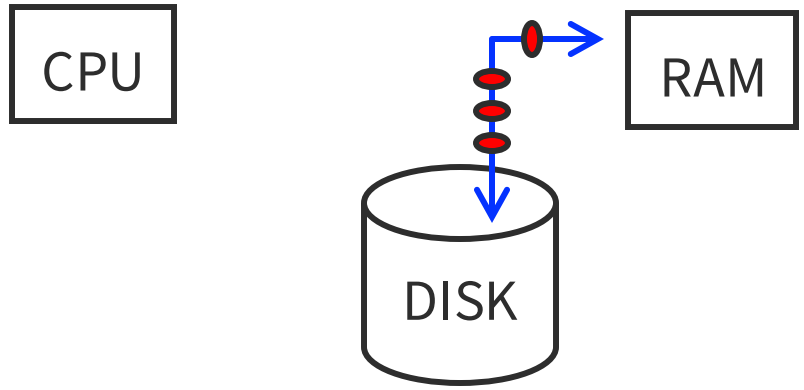
Programs: virtual addresses



DMA Issues (2): Virtual Mem

Issue #2: DMA meets *Paged* Virtual Memory

DMA destination page
may get swapped out



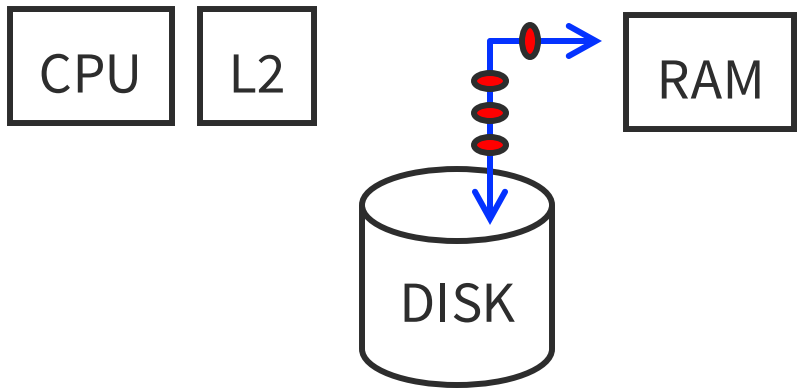
Solution: Pin the page before initiating DMA

- Alternate solution: Bounce Buffer
- DMA to a pinned kernel page, then memcpy elsewhere

DMA Issues (3): Caches

Issue #3: DMA meets Caching
DMA-related data could
be cached in L1/L2

- DMA to Mem: cache is now stale
- DMA from Mem: dev gets stale data



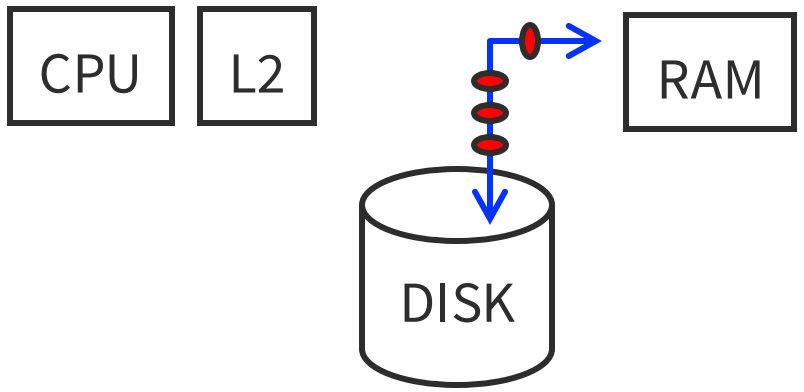
Solution: (software enforced coherence)

- OS flushes some/all cache before DMA begins
- Or: don't touch pages during DMA
- Or: mark pages as uncacheable in page table entries
- (needed for Memory Mapped I/O too!)

DMA Issues (3): Caches

Issue #3: DMA meets Caching
DMA-related data could
be cached in L1/L2

- DMA to Mem: cache is now stale
- DMA from Mem: dev gets stale data



Solution: (hardware coherence aka snooping)

- cache listens on bus, and conspires with RAM
- DMA to Mem: invalidate/update data seen on bus
- DMA from mem: cache services the request if possible, otherwise RAM services

Programmed I/O vs Memory Mapped I/O

Programmed I/O

- Requires special instructions
- Can require dedicated hardware interface to devices
- Protection enforced via kernel mode access to instructions
- Virtualization can be difficult

Memory-Mapped I/O

- Re-uses standard load/store instructions
- Re-uses standard memory hardware interface
- Protection enforced with normal memory protection scheme
- Virtualization enabled with normal memory virtualization scheme

Polling vs. Interrupts

How does program learn device is ready/done?

1. Polling: Periodically check I/O status register
   - Common in small, cheap, or real-time embedded systems
   + Predictable timing, inexpensive
   – Wastes CPU cycles
2. Interrupts: Device sends interrupt to CPU
   - Cause register identifies the interrupting device
   - Interrupt handler examines device, decides what to do
   + Only interrupt when device ready/done
   – Forced to save CPU context (PC, SP, registers, *etc.*)
   – Unpredictable, event arrival depends on other devices' activity

I/O Takeaways

Diverse I/O devices require hierarchical interconnect which is more recently transitioning to point-to-point topologies.

Memory-mapped I/O is an elegant technique to read/write device registers with standard load/stores.

Interrupt-based I/O avoids the wasted work in polling-based I/O and is usually more efficient.

Modern systems combine memory-mapped I/O, interrupt-based I/O, and direct-memory access to create sophisticated I/O device subsystems.